First Hit Fwd Refs
End of Result Set

Generate Collection Print

L45: Entry 2 of 2 File: USPT Jun 27, 2000

DOCUMENT-IDENTIFIER: US 6081665 A
TITLE: Method for efficient soft real-time execution of portable byte code computer programs

Abstract Text (1):
The invention is a method for use in executing portable virtual machine computer programs under real-time constraints. The invention includes a method for implementing a single abstract virtual machine execution stack with multiple independent stacks in order to improve the efficiency of distinguishing memory pointers from non-pointers. Further, the invention includes a method for rewriting certain of the virtual machine instructions into a new instruction set that more efficiently manipulates the multiple stacks. Additionally, using the multiple-stack technique to identify pointers on the run-time stack, the invention includes a method for performing efficient defragmenting real-time garbage collection using a mostly stationary technique. The invention also includes a method for efficiently mixing a combination of byte-code, native, and JIT-translated methods in the implementation of a particular task, where byte-code methods are represented in the instruction set of the virtual machine, native methods are written in a language like C and represented by native machine code, and JIT-translated methods result from automatic translation of byte-code methods into the native machine code of the host machine. Also included in the invention is a method to implement a real-time task dispatcher that supports arbitrary numbers of real-time task priorities given an underlying real-time operating system that supports at least three task priority levels. Finally, the invention includes a method to analyze and preconfigure virtual memory programs so that they can be stored in ROM memory prior to program.

Brief Summary Text (5):
class (object) declarations. The purpose of this requirement is to reduce conflicts in the global name space. Java provides standardized support for multiple threads (lightweight tasks) and automatic garbage collection of dynamically-allocated memory. Furthermore, Java fully specifies the behavior of every operator on every type, unlike C and C++ which leave many behaviors to be implementation dependent. These changes were designed to improve software scalability, reduce software development and maintenance costs, and to achieve full portability of Java software. Anecdotal evidence suggests that many former C and C++ programmers have welcomed these language improvements.

Brief Summary Text (6):
One distinguishing characteristic of Java is its execution model. Java programs are first translated into a fully portable standard byte code representation. The byte code is then available for execution on any Java virtual machine. A Java virtual machine is simply any software system that is capable of understanding and executing the standard Java byte code representation. Java virtual machine support is currently available for AIX, Apple Macintosh, HPUX, Linux, Microsoft NT, Microsoft Windows 3.1, Microsoft Windows 95, MVS, Silicon Graphics IRIX, and Sun Solaris. Ports to other environments are currently in progress. To prevent viruses from being introduced into a computer by a foreign Java byte-code program, the Java virtual machine includes a Java byte code analyzer that verifies the byte code does

not contain requests that would compromise the local system. By convention, this byte code analyzer is applied to every Java program before it is executed. Byte code analysis is combined with optional run-time restrictions on access to the local file system for even greater security. Current Java implementations use interpreters to execute the byte codes but future high-performance Java systems will have the capability of translating byte codes to native machine code on the fly. In theory, this will allow Java programs to run approximately at the same speed as C++.

Brief Summary Text (11):
PERC has much to offer developers of embedded real-time systems. High-level abstractions and availability of reusable software components shorten the time-to-market for innovative products. Its virtual machine execution model eliminates the need for complicated cross-compiler development systems, multiple platform version maintenance, and extensive rewriting and retesting each time the software is ported to a new host processor. It is important to recognize that the embedded computing market is quite large. Industry observers have predicted that by the year 2010, there will be ten times as many software programmers writing embedded systems applications as there will be working on software for general purpose computers.

Brief Summary Text (12):
Unlike many existing real-time systems, most of the applications for which PERC is intended are highly dynamic. New real-time workloads arrive continually and must be integrated into the existing workload. This requires dynamic management of memory and on-the-fly schedulability analysis. Price and performance issues are very important, making certain traditional real-time methodologies cost prohibitive. An additional complication is that an application developer is not able to test the software in each environment in which it is expected to run. The same Java byte-code application would have to run within the same real-time constraints on a 50 MHz 486 and on a 300 MHz Digital Alpha. Furthermore, each execution environment is likely to have a different mix of competing applications with which this code must contend for CPU and memory resources. Finally, every Java byte-code program is supposed to run on every Java virtual machine, even a virtual machine that is running as one of many tasks executing on a time-sharing host. Clearly, time-shared virtual machines are not able to offer the same real-time predictability as a specially designed PERC virtual machine embedded within a dedicated microprocessor environment. Nevertheless, such systems are able to provide soft-real-time response.

Brief Summary Text (14):
Accurate Garbage Collection, as the term is used in this invention disclosure, describes garbage collection techniques in which the garbage collector has complete knowledge of which memory locations hold pointers and which don't. This knowledge is necessary in order to defragment memory.

Brief Summary Text (15):
Byte code is a term of art that describes a method of encoding instructions (for interpretation by a virtual machine) as 8-bit numbers, each pattern of 8 bits representing a different instruction.

Brief Summary Text (16):
Conservative Garbage Collection, as the term is used in this invention disclosure, describes garbage collection techniques in which the garbage collector makes conservative estimates of which memory locations hold pointers. Conservatively, the garbage collector assumes that any memory location holding a valid pointer value (a legal memory address) contains a pointer. Fully conservative garbage collectors cannot defragment memory. However, partially conservative garbage collectors (in which some pointers are accurately identified) can partially defragment memory.

Brief Summary Text (19):

Defragmenting Garbage Collection, as the term is used in this invention disclosure, describes a garbage collection technique that relocates in-use memory objects to contiguous locations so as to coalesce multiple segments of free memory into larger free segments.

Brief Summary Text (22):
Garbage Collection is a term of art describing the automatic process of discovering regions of computer memory that were once allocated to a particular purpose but are no longer needed for that purpose and reclaiming said memory to make it available for other purposes.

Brief Summary Text (23):
Garbage Collection Flip, as the term is used in this invention disclosure, is the process of beginning another pass of the garbage collector. When garbage collection begins, the roles assigned to particular memory regions exchange; thus the use of the term "flip."

Brief Summary Text (24):
Heap is a term of art describing a region of memory within which arbitrary sized objects can be allocated and deallocated to satisfy the dynamic memory needs of application programs.

Brief Summary Text (26):
Java, a trademark of Sun Microsystems, Inc., is an object-oriented programming language with syntax derived from C and C++, which provides automatic garbage collection and multi-threading support as part of the standard language definition.

Brief Summary Text (27):
JIT, as the term is used in this invention disclosure, is an acronym standing for "just in time." The term is used to describe a system for translating Java byte codes to native machine language on the fly, just-in-time for its execution. We consider any translation of byte code to machine language which is carried out by the virtual machine to be a form of JIT compilation.

Brief Summary Text (33):
Read Barrier is a term of art describing a special check performed each time application code fetches a value from a heap memory location. The read barrier serves to coordinate application processing with garbage collection.

Brief Summary Text (35):
Real-Time Garbage Collection, as the term is used in this invention disclosure, describes a garbage collection technique that allows incremental interleaved execution of garbage collection and application code which is organized such that high-priority application code can preempt the garbage collector when necessary and garbage collection is consistently provided with adequate execution time to allow it to make guaranteed forward progress at a rate sufficient to satisfy the allocation needs of real-time application programs.

Brief Summary Text (36):
Root Pointer is a term of art describing a pointer residing outside the heap which may point to an object residing within the heap. The garbage collector considers all objects reachable through some chain of pointers originating with a root pointer to be "live."

Brief Summary Text (37):
RTVMM, as the term is used in this invention disclosure, is an acronym standing for Real-Time Virtual Machine Method. This acronym represents the invention disclosed by this document.

Brief Summary Text (40):
Slow Pointer is a term specific to this invention disclosure which describes pointers that are implemented in such a way that they provide coordination with a background garbage collection task. Various implementations of slow pointers are possible. In general, fetching, storing, and indirecting through slow pointer variables is slower than performing the same operation on fast pointer variables.

Brief Summary Text (42):
Tending is a term of art describing the garbage collection process of examining a pointer to determine that the object it refers to is live and arranging for the referenced object to be subsequently scanned in order to tend all of the pointers contained therein.

Brief Summary Text (44):
Virtual Machine is a term of art that describes a software system that is capable of interpreting the instructions encoded as numbers according to a particular agreed upon convention.

Brief Summary Text (45):
Write Barrier is a term of art describing a special check performed each time application code stores a value to a heap memory location. The write barrier serves to coordinate application processing with garbage collection.

Brief Summary Text (47):
The invention is a real-time virtual machine method (RTVMM) for use in implementing portable real-time systems. The RTVMM provides efficient support for execution of portable byte-code representations of computer programs, including support for accurate defragmenting real-time garbage collection. Efficiency is measured both in terms of memory utilization, CPU time, and programmer productivity. Programmer productivity is enhanced through reduction of the human effort required to make the RTVMM available in multiple execution environments.

Brief Summary Text (48):
The innovations comprised in this disclosure include the following:

Brief Summary Text (50):
2. A mechanism to translate traditional Java byte codes into the extended PERC byte codes at run-time, as new Java byte codes are loaded into the virtual machine's execution environment.

Brief Summary Text (53):
4. A set of C macros and functions that characterize the native-method application programmer interface (API). This API abstracts the native-method programmer's interface to the internal data structures, the run-time task scheduler, and the garbage collector.

Brief Summary Text (54):
5. A method for implementing mostly stationary defragmenting real-time garbage collection in software.

Drawing Description Text (3):
FIG. 2 illustrates the header information attached to each dynamically allocated memory object for purposes of performing garbage collection. These header fields consist of Scan-List, Indirect-Pointer, Activity-Pointer, Signature-Pointer, and optional Finalize-Link pointers.

Drawing Description Text (5):
FIG. 4 illustrates from-space and to-space regions, in which three live objects are being copied out of from-space into to-space. In this illustration, objects B and C have already been copied and object A is scheduled for copying. Objects D and E

were presumably copied into to-space by a previous garbage collection pass and object F was allocated from to-space during a previous garbage collection pass.

Drawing Description Text (6):
FIG. 5 illustrates the appearance of the pointer and non-pointer stack activation frames immediately before calling and immediately following entry into the body of a Java method. The stacks are assumed to grow downward. In preparation for the call, arguments are pushed onto the stack. Within the called method, the frame pointer (fp) is adjusted to point at the memory immediately above the first pushed argument and the stack pointer (sp) is adjusted to make room for local variables to be stored on the stack.

Drawing Description Text (18):
FIG. 17 provides the C declaration of the structure used internal to the PERC implementation to represent a raw class file that has been read into memory. The class-file loader analyzes this object to create an appropriate Class representation.

Drawing Description Text (22):
FIG. 21 provides the C declaration of the structure used internal to the PERC implementation to represent a HashCache structure. Each HashCache structure is capable of representing three recycled hash values. HashCache structures are generally created by modifying the signature field of existing HashLock structures during garbage collection.

Drawing Description Text (29):
FIG. 28 provides a C macro definition of the SetJmp( ) macro, which is a version of the standard C setjmp( ) function specialized for the PERC virtual machine execution environment.

Drawing Description Text (30):
FIG. 29 provides a C macro definition of the UnsetJmp( ) macro, which is used within the PERC virtual machine execution environment to replace the current exception handling context with the surrounding exception handling context.

Drawing Description Text (31):
FIG. 30 provides a C macro definition of the LongJmpo macro, which is a version of the standard C longjmp( ) function specialized for the PERC virtual machine execution environment. Note that this macro makes use of perclongjmp( ) whose implementation is not provided. perclongjmp( ) expects as parameters a representation of the machine's registers including its instruction pointer, the value of the pointer stack pointer, the value of the non-pointer stack pointer, and the return value to be returned to the point of the JIT version of setjmp( ).

Drawing Description Text (33):
FIG. 32 illustrates the signature structure used to represent the memory layout of heap-allocted objects. total.sub.-- length is the total number of words comprising the object, excluding the object's header words, but including its signature if the signature happens to be appended to the end of the data. All pointers are assumed to be word aligned within the structure. Use last.sub.-- descriptor to symbolically represent the word offset of the last word within the corresponding object that might contain a pointer. When the garbage collector scans the corresponding object in search of pointers, it looks no further than the word numbered last.sub.-- descriptor. type.sub.-- code comprises a 2-bit type tag in its most significant bits, with the remaining 30 bits representing the value of last.sub.-- descriptor. bitmap is an array of integers with each integer representing 32 words of the corresponding object, so there are a total of ceiling(last.sub.-- descriptor/32) entries in the array. (bitmap[0]&0.times.01), which represents the first word of the corresponding object, has value 1 if and only if the first word is a pointer.

Drawing Description Text (39):
FIG. 38 provides C declarations of the standard garbage collection header and accompanying macros for manipulation and access to the header information

Drawing Description Text (42):
FIG. 41 provides C macros to enable the reading and writing of memory representing the fields of heap-allocated structures.

Drawing Description Text (67):
FIG. 66 provides the definition of a C macro used within the implementation of the PERC virtual machine to support preemption of the currently executing thread.

Drawing Description Text (68):
FIG. 67 provides the definitions of C macros for saving and restoring the state of the PERC virtual machine surrounding each preemption point.

Drawing Description Text (69):
FIG. 68 provides the C implementation of the PERC virtual machine, except that cases to handle each byte code are excluded.

Drawing Description Text (70):
FIG. 69 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IADD instruction, which adds the two integers on the top of the Java stack, placing the result on the Java stack.

Drawing Description Text (71):
FIG. 70 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the AASTORE instruction.

Drawing Description Text (72):
FIG. 71 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the FCMPL instruction.

Drawing Description Text (73):
FIG. 72 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IFEQ instruction.

Drawing Description Text (74):
FIG. 73 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the JSR instruction.

Drawing Description Text (75):
FIG. 74 provides the C code to be inserted into the PERC virtual machine

Drawing Description Text (77):
FIG. 75 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the TABLESWITCH instruction.

Drawing Description Text (78):
FIG. 76 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the LOOKUPSWITCH instruction.

Drawing Description Text (79):
FIG. 77 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the IRETURN instruction.

Drawing Description Text (80):
FIG. 78 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the GETSTATIC.sub.-- QNP8 instruction.

Drawing Description Text (81):
FIG. 79 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the PUTFIELD.sub.-- Q instruction.

Drawing Description Text (82):
FIG. 80 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKEVIRTUAL.sub.-- FQ instruction.

Drawing Description Text (83):
FIG. 81 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKESPECIAL.sub.-- Q instruction.

Drawing Description Text (84):
FIG. 82 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKESTATIC.sub.-- Q instruction.

Drawing Description Text (85):
FIG. 83 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INVOKEINTERFACE.sub.-- Q instruction.

Drawing Description Text (86):
FIG. 84 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the NEW.sub.-- Q instruction.

Drawing Description Text (87):
FIG. 85 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the NEWARRAY instruction. FIG. 86 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the ANEWARRAY.sub.-- Q instruction.

Drawing Description Text (88):
FIG. 87 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the ATHROW instruction.

Drawing Description Text (89):
FIG. 88 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the CHECKCAST.sub.-- Q instruction.

Drawing Description Text (90):
FIG. 89 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the INSTANCEOF.sub.-- Q instruction.

Drawing Description Text (91):
FIG. 90 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the MONITORENTER instruction.

Drawing Description Text (92):
FIG. 91 provides the C code to be inserted into the PERC virtual machine template illustrated in FIG. 68 in order to implement the MONITOREXIT instruction.

Drawing Description Text (96):
FIG. 95 illustrates the organization of free lists, partitioned by region, but combined into a single global pool to support efficient constant-time allocation. In this figure, the three regions (indicated by the three large objects on the left side of the figure) are prioritized such that preference is given to allocating

h e b b g e e e f c e e ge

from the top region first followed by the middle region and then the bottom region. This figure illustrates only two size categories, 16 and 32. In the actual implementation, there are free lists for each size category, ranging from size 4 to size 512K.

Detailed Description Text (3):
The PERC virtual machine consists primarily of an interpreter for the PERC byte-code instruction set, a task (thread) dispatcher, and a garbage collector written in C which runs as an independent real-time task. Most of the functionality of the PERC execution environment is provided by standard library and system programs that accompany the virtual machine and are executed by the virtual machine.

Detailed Description Text (12):
Methods represented as byte codes are interpreted by the PERC virtual machine. The interpreter, known throughout this invention disclosure as pvm() (for PERC virtual machine), uses three stacks for execution: (1) the traditional C stack, (2) an explicitly managed stack for representation of PERC pointer values, and (3) an explicitly managed stack for representation of PERC non-pointer values. The C stack holds C-declared local variables and run-time state information associated with compiler generated temporaries. The PERC pointer stack holds the pointer arguments passed as inputs to the method, pointer local variables, temporary pointers pushed during expression evaluation, and pointer values pushed as arguments to methods called by the current method. The PERC non-pointer stack holds non-pointer arguments passed as inputs to the method, non-pointer local variables, temporary non-pointer values pushed during expression evaluation, and non-pointer values pushed as arguments to be called by this method. The pointer and non-pointer stack activation frames are illustrated in FIG. 5 and FIG. 6.

Detailed Description Text (16):
Native methods use the same three stacks as are used by the PERC virtual machine to execute byte-code methods.

Detailed Description Text (18):
PERC, like Java, supports four distinct forms of method invocation. These are known as (1) virtual, (2) special (non-virtual), (3) static, and (4) interface. With virtual and special method invocations, there is an implicit (not seen by the Java programmer) "this" argument passed to the called method. The "this" argument refers to the object on which the called method will operate. The distinctions between these different method invocations are described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley.

Detailed Description Text (20):
The PERC implementation represents every PERC object with a data structure patterned after the templates provided in FIG. 15, FIG. 16, and FIG. 24. In all of these structures, the second field is a pointer to a MethodTable data structure (see FIG. 23). The PERC execution environment maintains one MethodTable data structure for each defined object type. All instantiated objects of this type point to this shared single copy. The jit.sub.-- interfaces array field of the MethodTable structure has one entry for each virtual method supported by objects of this type. The mapping from method name and signature to index position is defined by the class loader, as described in "The Java Virtual Machine Specification", by Lindholm and Yellin, 1996, Addison-Wesley. To execute the JIT version of a PERC method using a virtual method lookup, branch to the code represented by jit.sub.-- interfaces[method.sub.-- index]. Normally, the JIT version of the byte code will only be invoked directly from within another JIT-compiled method. If a native or untranslated byte-code method desires to invoke another method using virtual method lookup, the search for the target method generally proceeds differently. First, we find the target object's MethodTable data structure (as above) and then follow the methods pointer to obtain an array of pointers to Method objects. Within the Method object, we consult the access.sub.-- flags field to determine if the target method

is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be implemented by byte codes. See FIG. 49, FIG. 45, and FIG. 46.

Detailed Description Text (27):
When a method is invoked through an interface declaration, the called method's name and signature is stored as part of the calling method's code representation. The compiler ensures that the object to be operated on has a method of the specified name and signature. However, it is not possible to determine prior to run time the index position within the method table that holds the target method. Thus it is necessary to examine the target object's mtable field, which points to the corresponding MethodTable structure. We follow the MethodTable's methods pointer to find an array of pointers to Method structures. And we search this array for a method that matches the desired name and signature. Once found, we invoke this method. We examine the Method object that represents the target procedure and consult its access.sub.-- flags field to determine if the method is represented by native code (ACC.sub.-- NATIVE) or JIT translation of byte code (ACC.sub.-- JIT). If neither of these flags is set, the method is assumed to be represented as byte code. See FIG. 50, FIG. 51, FIG. 45, and FIG. 46.

Detailed Description Text (59):
When JIT-translated code invokes a method that is implemented by Java byte code, it is necesary to switch the execution protocol prior to invoking pvm(). Rather than requiring JIT-generated code to check whether this protocol switch is necessary prior to each method invocation, we provide each byte-code method with a stub procedure that honors the JIT execution protocol. This stub procedure switches from JIT to C protocols and then invokes pvm() with appropriate arguments. In more detail, the stub procedure performs the following:

Detailed Description Text (65):
6. For coordination with the garbage collector, we keep track of how high the pointer stack has grown during the current execution time slice. Since the stack grows downward, the high-water mark is represented by the minimum value of .sub.-- psp.

Detailed Description Text (70):
11. For coordination with the garbage collector, we keep track of how low the pointer stack has shrunk during the current execution time slice. Since the stack grows downward, the low-water mark is represented by the maximum value of .sub.-- pfp.

Detailed Description Text (75):
5.0 Disciplines to Support Accurate Real-Time Garbage Collection

Detailed Description Text (76):
To support real-time performance, garbage collection runs asynchronously, meaning that the garbage collection thread interleaves with application code in arbitrary order. To support accurate garbage collection, it is necessary for the garbage collector to always be able to distinguish a thread's pointer variables (including stack-allocated variables and variables held in machine registers) from the thread's non-pointer variables.

Detailed Description Text (77):
To require each thread to maintain all pointers in variables that are at all times easily identifiable by the garbage collector imposes too great an overhead on overall performance. Thus, the PERC virtual machine described in this invention disclosure implements the following compromises:

Detailed Description Text (79):
2. Between preemption points, the thread is allowed to hold pointers in variables

that may not be visible to the garbage collector. In this disclosure, we characterize such variables as "fast pointers." Fast pointers are typically declared in C as local variables, and may be represented either by machine registers or slots on the C stack.

Detailed Description Text (80):
3. Pointer variables that are visible to the garbage collector are known throughout this disclosure as "slow pointers". Slow pointers are typically represented by locations on the PERC pointer stack and by certain C-declared global variables identified as "root pointers".

Detailed Description Text (81):
4. Immediately following each preemption, the thread must consider all of its fast pointers to be invalid. In preparation for each preemption, the thread must copy the values of essential fast-pointer variables into slow pointers. Following each preemption, essential fast pointers are restored from the values previously stored in slow pointer variables. Note that, while the thread was preempted, a drefragmenting garbage collector might have relocated particular objects, requiring certain pointer values to be modified to reflect the corresponding objects' new locations.

Detailed Description Text (82):
5. Each C function in the virtual machine implementation is identified as either preemptible or non-preemptible. Before calling a preemptible function, the caller must copy all of its essential fast pointers into slow pointers. When the called function returns, the caller must restore the values of these fast-pointer variables by copying from the slow-pointer variables in which their values were previously stored. Throughout this disclosure, we refer to preemptible functions as "slow functions" and to non-preemptible functions as "fast functions."

Detailed Description Text (85):
Restrictions. In order to coordinate application processing with garbage collection, it is necessary for authors of native methods and other C libraries to avoid certain "legal" C practices:

Detailed Description Text (88):
3. Do not store tag information (e.g. low-order bits of a word) within memory locations that are identified as pointers to the garbage collector.

Detailed Description Text (89):
4. Do not store pointers to the C static region or to arbitrary derived addresses in locations identified as garbage-collected pointers, except that pointers to objects residing in the ROMized region are allowed. Note: A derived address is a location contained within an object. The garbage collector assumes all pointers refer to the base or beginning address of the referenced object.

Detailed Description Text (90):
5. Do not directly access the fields contained within heap objects. Instead, use the GetHeapPtr(), GetHeapNonPtr(), SetHeapPtr(), and SetHeapNonPtr() macros.

Detailed Description Text (91):
6. When declaring fields and variables that point to the C static region, identify such fields as non-pointers insofar as garbage collection is concerned.

Detailed Description Text (92):
7. Pointers to garbage-collected objects cannot be stored in the C static region unless such pointers have been registered as root pointers.

Detailed Description Text (93):
5.1 Access to Heap Objects

Detailed Description Text (94):
It is necessary to provide parameterized access to heap memory so as to facilitate the implementation of read and write barriers. The following macros serve to copy variables between different kinds of representations. See FIG. 41 for implementations of these macros.

Detailed Description Text (96):
Given that (base) (field.sub.-- expr) is an expression representing a pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, and that field.sub.-- type represents the type of (base) (field.sub.-- expr), return the fast pointer that represents this heap pointer's value. In the process, we may have to "tend" the pointer's value. Optionally, we may overwrite in place the value of (base) (field.sub.-- expr), so this expression should be a C 1-value. Sample usage:

Detailed Description Text (99):
Given that (base) (field.sub.-- expr) is an expression representing a pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, that field.sub.-- type represents the type of (base) (field.sub.-- expr), and that field.sub.-- value is also of type field.sub.-- type, assign field.sub.-- value to (base) (field.sub.-- expr). In the process, we may have to "tend" field.sub.-- value. Note that (base) (field.sub.-- expr) must be a C 1-value. Sample usage: SetHeapPtr(.sub.-- current.sub.-- thread, .fwdarw.current.sub.-- exception, Thread *, Object *, new.sub.-- exception);

Detailed Description Text (101):
Given that (base) (field.sub.-- expr) is an expression representing a non-pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, and that field.sub.-- type represents the type of (base) (field.sub.-- expr), return the non-pointer value that represents this heap location's value. Sample usage:

Detailed Description Text (104):
Given that (base) (field.sub.-- expr) is an expression representing a non-pointer residing within a heap-allocated object, that base.sub.-- type represents the type of base, that field.sub.-- type represents the type of (base) (field.sub.-- expr), and that field.sub.-- value is also of type field.sub.-- type, assign field.sub.-- value to (base) (field.sub.-- expr). Note that (base) (field.sub.-- expr) must be a C 1-value. Sample usage:

Detailed Description Text (108):
In this code, base is assumed to point to the beginning of a heap-allocated object which is declared to be an array of integers (the type of base should be specified by the third argument of GetHeapInArrayNonPtr()). The second argument is combined with the first to obtain the object whose address represents the location at which the longlong integer will be fetched. There are four different macros provided for this sort of access to array data:

Detailed Description Text (113):
The following help macro is intended to facilitate the use of garbage collection macros in application code. See FIG. 39 for the implementation of this macro. int SameObjects(void *p1, void *p2)

Detailed Description Text (114):
With certain garbage collection techniques, it is possible that two fast pointer objects refer to the same object even though their pointer values are different. This might occur, for example, if an object is being copied in order to compact live memory and one pointer refers to the original location of the object and the other pointer refers to the new copy of the object. Programmers should use the

SameObject() macro to compare fast pointers for equality. This macro returns non-zero if and only if its two pointer arguments refer to the same object.

Detailed Description Text (116):
The following macros are used to access and manipulate string data and slice objects. char *GetStringData(String *s)

Detailed Description Text (125):
Additionally, we provide the following C functions, which can be assumed to be preemptible unless fast appears in their names or they are specifically being described as not preemptible, for manipulation of string data. Programmers who invoke the fast functions below should take care to avoid passing arguments that represent "long" strings, since doing so would increase preemption latency.

Detailed Description Text (127):
Compares the data of string1 to that of string2, returning 0 if the strings are equal, -1 if string1 lexicographically precedes string2, and 1 if string1 lexicographically follows string2.

Detailed Description Text (135):
Given that jstring represents a String object, returns a null-terminated array of characters representing the data of jstring. The null-terminated array is heap allocated.

Detailed Description Text (143):
Native methods and other C functions run fastest if they avoid frequent copying of values between local variables (stored on the PERC pointer stacks) and C-declared fast pointers. But pointers stored in C-declared variables are not necessarily preserved across preemption of the thread. Thus, it is necessary for the application code to copy from C-declared variables to macro-declared variables before each preemption. The following macros are used to manipulate the pointer stack.

Detailed Description Text (160):
The non-pointer stack holds integers, 8-byte long integers, floating point values, and 8-byte double-precision floating point values. The following macros are suggested for manipulation of the non-pointer stack.

Detailed Description Text (213):
Strict partitioning between fast and slow pointers, and requiring all heap memory access to be directed by way of heap access macros imposes a high overhead. Certain data structures are accessed so frequently that the PERC implementation treats them as special cases in order to improve system performance. In particular, the following exceptions are supported:

Detailed Description Text (214):
1. Note that the PERC stacks dedicated to representation of pointer and non-pointer data respectively are heap allocated. According to the protocols described above, every access to PERC stack data should be directed by way of heap access macros. Since stack operations are so frequent, we allow direct access to stack data using traditional C pointer indirection. This depends on the following:

Detailed Description Text (215):
a. The stack pointers are represented by C global variables declared as pointers. Access to stack data uses C pointer indirection, without enforcement of special read or write barriers. (See FIG. 64 and FIG. 65)

Detailed Description Text (218):
d. During execution of a time slice, the thread's pointer stack is assumed to hold fast pointers. However, when the thread is preempted, the garbage collector needs

to see the stack object's contents as slow pointers. When the thread is preempted, the dispatcher scans that portion of the stack that has been modified during the current time slice in order to convert all fast pointers to slow pointers (See FIG. 53 and FIG. 44). We maintain a low-water mark representing a bound on the range of stack memory that has been impacted by execution of the task during its current time slice to reduce the need for redundant stack scanning.

Detailed Description Text (219):
2. When the pvm() (PERC Virtual Machine byte code interpreter) is executing byte-code methods, the method's byte code is represented by a string of bytes. The byte-code instructions are stored in heap memory, suggesting that every instruction fetch needs to incur the overhead of a heap-access macro. To improve the performance of instruction fetching, we allow instruction fetching to bypass the standard heap access macro. Doing so depends on the following:

Detailed Description Text (223):
3. During interpretation of byte-code methods, the constant pool is frequently accessed. Rather than incurring the overhead of a standard heap access macro, we obtain a trustworthy C pointer to the constant pool data structure and refer directly to its contents. For this purpose, we use the GetCP() macro (See FIG. 40). C subscripting expressions based on the value returned by GetCP() are considered valid up to the time at which the thread is next preempted. Following each preemption, the pointer must be recomputed through another application of the GetCP () macro.

Detailed Description Text (230):
The PERC virtual machine. The PERC interpreter (virtual machine) is invoked once for each method to be interpreted. If the method to be interpreted contains synchronized or try blocks, a jump buffer is initialized according to the same protocol described above.

Detailed Description Text (233):
1. The virtual machine implementation: pvm() deserves special treatment since its performance is so critical. The caller of pvm(), which may be a stub procedure for a particular byte-code method, must set up the PERC stack activation frames for execution of this pvm(). Upon return from pvm( ), the caller removes the activation frames from the stacks. In place of the local arguments, the caller leaves a single placeholder to represent the return value on whichever PERC stack is appropriate, or leaves no placeholder if the method is declared as returning void. The activation frame maintenance performed by a stub procedure is described in Section 4.2 on page 18. The activation frame maintenance performed by an invokeVirtual(), invokeSpecial(), invokestatico, or invokeInterface() function is described later in this section under subheadings "PrepareJavaFrames()" and "ReclaimFrames()".

Detailed Description Text (235):
2. Native methods: Like the virtual machine, each invocation of a native method must be preceded by the preparation of PERC stack activation frames. The format of the activation frames and the protocol for allocation of local pointers is exactly the same for native methods as for pvm().

Detailed Description Text (236):
3. Fast procedures: A fast procedure is a C function called by the pvm(), native methods, or other fast or slow procedures, that is by design, not preemptible. Arguments to a fast procedure are passed on the C stack using traditional C argument passing conventions. Fast procedures should not attempt to access information placed on the PERC stacks by the calling context. (The current PERC implementation makes a non-portable exception to this rule in the implementation of the FastInvokeo macros described in Section 2.0.) This is because the code generated by a custom C compiler that is designed to support accurate garbage collection of C code may place information onto the PERC stacks that would obscure

the data placed there by outer contexts.

Detailed Description Text (242):
AllocLocalPointers(). The AllocLocalPointers() macro may be used only within the implementations of the pvm() and of native methods. If present, the AllocLocalPointers() macro must follow the last local declaration and precede the first line of executable code. The parameterization is as follows:

Detailed Description Text (244):
To find the offset of the top-of-stack entry within an activation frame that has no local pointers, use the following:

Detailed Description Text (246):
BuildFrames(). The BuildFrames() macro is required in each slow procedure. This macro must follow the last local declaration and must precede the first line of executable code. The parameterization is as follows:

Detailed Description Text (271):
AdjustLowWaterMark. Both ReclaimFrames() and DestroyFrames() make use of the AdjustLowWaterMark() macro, which is defined in FIG. 55. The purpose of this macro is to keep track of the lowest point to which the pointer stack has shrunk during execution of the current time slice. When this task is preempted, all of the pointers between the low-water mark and the current top-of-stack pointer are tended. By tending these pointers at preemption time, it is not necessary to enforce the normal write barrier with each update to the pointer stack.

Detailed Description Text (272):
6.0 The PERC Virtual Machine

Detailed Description Text (273):
The PERC Virtual Machine describes the C function that interprets Java byte codes. This C function, illustrated in FIG. 68, is named pvm(). The single argument to pvm () is a pointer to a Method structure, which includes a pointer to the byte-code that represents the method's functionality. Each invocation of pvm() executes only a single method. To call another byte-code method, pvm() recursively calls itself. Note that pvm() is reentrant. When multiple Java threads are executing, each thread executes byte-code methods by invoking pvm() on the thread's run-time stack.

Detailed Description Text (276):
When an exception is caught, pvm() searches for the appropriate handler in its exception-handling table. This search proceeds as follows:

Detailed Description Text (287):
The JSR instruction (See FIG. 73) jumps to a subroutine by branching to the byte-code instruction obtained by adding the two-byte signed quantity which is part of the instruction encoding to the current value of the program counter and pushing the return address onto the non-pointer stack. Note that the return address is represented as the integer offset within the current method's byte code rather than an actual pointer. This is because the garbage collector does not deal well with pointers that refer to internal addresses within objects rather than to the objects' starting addresses. Note also that the JSR instruction also invokes the PVMPremptionPoint() macro.

Detailed Description Text (306):
Note that the PERC virtual machine maintains several stacks. Thus, it is necessary to augment the traditional C jump buffer data structure with the additional fields necessary to represent this information. The supplementary information includes:

Detailed Description Text (311):
The SetJmp() macro (See FIG. 28) initializes appropriate fields of the

PERCEnvironment data structure and then calls the C setjmp() function. UnsetJmp() (See FIG. 29) has the effect of removing the most recently established exception handling context. Following execution of UnsetJmp(), whatever exception handling context had been active at the moment this context was established once again becomes the active context. The LongJmp() macro (See FIG. 30) takes responsibility for calling longjmp() in addition to setting other state variables as appropriate. The throwException() function (See FIG. 92) invokes LongJmp(), but only after first verifying that an exception handling context exists. If there is no current exception handling context, throwException() calls the topLevelExceptionHandler() routine.

Detailed Description Text (313):
In concept, every Java object has an associated lock and an associated hash value. However, in practice, the large majority of Java objects never make use of either the lock or the hash value. Note that in systems that never relocate objects, converting an object's address to an integer value is probably the easiest way to obtain a hash value. However, in systems that make use of defragmenting garbage collectors, such as in the PERC execution environment, it is necessary to use some other technique to represent hash values.

Detailed Description Text (314):
In the PERC implementation, every object has a HashLock pointer field, which is initialized to NULL. When either a lock or a hash value is needed for the object, a HashLock object (see FIG. 20) is allocated and initialized, and the HashLock pointer field is made to refer to this HashLock object. Note that each HashLock object has the following fields:

Detailed Description Text (320):
In determining the next available hash value, hashCode() first consults its list of previously assigned hash values for which the corresponding objects have been reclaimed by the garbage collector. (Once an object has been reclaimed by the garbage collector, its hash value can be reused.) If this list is non-empty, hashCode() assigns one of these hash values. Otherwise, it increments a static counter and uses its incremented value as the new hash value.

Detailed Description Text (321):
Obtaining and releasing monitor locks. When application code desires to enter a monitor, it executes the enterMonitor instruction. This instruction first consults the object's lock field. If this field is NULL, it allocates a HashLock object, initializes its count field to 1, sets its u.owner field to represent the current thread, and grants access to the newly locked object. If the lock field is non-NULL, enterMonitor examines the contents of the HashLock object to determine whether access to the lock can be granted. If the count field equals 0, or if the u.owner field refers to the currently executing thread, the count field is incremented, the u.owner field is made to point to the current thread if it doesn't already, and access is granted to the newly locked object. Otherwise, this lock is owned by another thread. The current thread is placed onto the waiting.sub.-- list queue and its execution is blocked until the object's lock can be granted to this thread. Priority inheritance describes the notion that if a high-priority thread is forced to block waiting for a low-priority thread to release its lock on a particular object, the low-priority thread should temporarily inherit the priority of the higher priority blocked task. This is because, under this circumstance, the urgency of the locking task is increased by the fact that a high-priority task needs this task to get out of its way. The PERC virtual machine implements priority inheritance. Furthermore, the waiting.sub.-- list queue is maintained in priority order.

Detailed Description Text (322):
When a thread leaves a monitor, it releases the corresponding lock. This consists of the following steps:

h e b b g e e e f c e e ge

Detailed Description Text (324):
2. Decrementing the count field. If the new value of count, following the decrement operation, is non-zero, this is all that must be done. Otherwise, continue by executing the steps that follow.

Detailed Description Text (327):
5. If the HashLock object's hash value field is non-zero, we must retain this HashLock object. In this case, we're done. Otherwise, continue by executing the following step.

Detailed Description Text (333):
We say that the C stack segments contain no pointers, but this is not entirely true. Since the C activation frame contains return addresses, the stack contains pointers to code. And since the activation frame includes saved registers, it probably contains the saved values of frame and stack pointers. To avoid the complications and efficiency hits that would be associated with the handling of these pointers by a relocating garbage collector, we require stack segments to be non-moving, except for one exception which is discussed below.

Detailed Description Text (334):
The C stack may also contain pointers to heap objects which were saved in registers or local variables within particular activation frames. The usage protocol requires that such variables be treated as dead insofar as the garbage collector is concerned.

Detailed Description Text (339):
In general, the PERC virtual machine is intended to support many more priority levels than might be supported by an underlying operating system. Further, the design of the real-time application programmer interface (API) is such that task dispatching cannot be fully relegated to traditional fixed priority dispatchers. Thus, the PERC virtual machine implements its own task dispatcher which communicates with an underlying thread model. To support this architecture, we use three priority levels, as follows:

Detailed Description Text (340):
1. At the highest priority, we run the task dispatcher. Most of the time, this thread is sleeping. However, it may be triggered by one of the following:

Detailed Description Text (353):
When the TaskDispatcher object is instantiated, the constructor invokes the initDispatcher() native method, illustrated in FIG. 54. This invention disclosure describes the implementation for the Microsoft Windows WIN32 API. The initDispatcher() method performs the following:

Detailed Description Text (354):
1. Registers as root pointers .sub.-- gc.sub.-- thread and .sub.-- dispatcher.sub.-- thread. These static variables identify the Thread objects that govern the garbage collection thread and the real-time dispatcher thread respectively.

Detailed Description Text (362):
When the TaskDispatcher's run method is invoked (automatically by the PERC run-time system since TaskDispatcher extends Thread), we perform the following:

Detailed Description Text (364):
2. startDispatcher() returns as a Java integer a Boolean flag which indicates whether garbage collection is enabled. In normal operation, garbage collection is always enabled. However, the system supports an option of disabling garbage collection so as to facilitate certain kinds of debugging and performance monitoring analyses.

Detailed Description Text (366):
a. Check nrt.sub.-- ready.sub.-- q to determine if all non-demon threads have terminated. If so, we shut the virtual machine down.

Detailed Description Text (369):
Note that implementation of task priorities is provided by the nrt.sub.-- ready.sub.-- q object. Its getNextThread() method always returns the highest priority thread that is ready to run. Note also that it would be straightforward to modify this code so that the duration of each thread's time slice is variable. Some thread's might require CPU time slices that are longer than 25 ms and others might tolerate time slices that are shorter. runThread() (See FIG. 54) performs the following:

Detailed Description Text (378):
7. If WaitForSingleObject() was timed out, the dispatcher's next responsibility is to preempt the currently executing task. If the task is currently running JIT code and it is not in the body of an atomic statement, it is already in a preemptible state. In other cases, preemption must be delayed until the task reaches a point outside of atomic statements at which garbage collection would be valid. The protocol consists of:

Detailed Description Text (389):
When a new thread is created, the system allocates a C stack, a PERC non-pointer stack, and a PFRC pointer stack. The size of the C stack is determined as a run-time option (specified on the command line if the virtual machine is running in a traditional desktop computing environment). The size of the PERC pointer and non-pointer stacks is specified by compile-time macro definitions, defined to equal 1024 words per stack.

Detailed Description Text (397):
So-called fast pointers refer directly to the corresponding memory objects using traditional C syntax. Fast pointers are stored on the traditional C stack or in machine registers. They are not scanned by the garbage collector. Thus, it is very important to make sure that garbage collection occurs at times that are coordinated with execution of application threads. (If the garbage collector were to relocate an object "while" an application thread was accessing that object by way of a fast pointer, the application thread would become confused.) Each application thread is responsible for periodically checking whether the system desires to preempt it. The following macro serves this purpose:

Detailed Description Text (400):
The typical usage of CheckPreemption() is illustrated in the following code fragment (See FIG. 43 for the implementation of the PreemptTask() macro):

Detailed Description Text (405):
Note that each time we preempt a task, we must be prepared to save and restore all of the fast pointers that are currently in use. However, in cases where a particular pointer variable is known to have been saved to the stack already, and has not been modified since it was last saved to the stack, it is possible to omit the save operation. The purpose of copying fast pointers into "local" pointer variable slots is to make them visible to the garbage collector. After the task has been preempted, the application task's fast pointers may no longer be valid. Thus, the application task must restore the fast-pointer variables by copying their updated values from the local pointer variables.

Detailed Description Text (419):
read(fd, buf, 128); //Note that buf must refer to static (non-relocatable) memory

Detailed Description Text (431):

Native libraries are implemented according to a protocol that allows references to dynamic objects to be automatically updated whenever the dynamic object is relocated by the garbage collector. However, if these native libraries call system routines which do not follow the native-library protocols, then the system routines are likely to become confused when the corresponding objects are moved. To avoid this problem, programmers who need to pass heap pointers to system libraries must make a stable copy of the heap object and pass a pointer to the stable copy. The stable copy should be allocated on the C stack, as a local variable. If necessary, upon return from the system library, the contents of the stable copy should be copied back into the heap. Note that on uniprocessor systems a non-portable performance optimization to this strategy is possible when invoking system libraries that are known not to block if thread preemption is under PERC's control. In particular, we can pass the system library a pointer to the dynamic object and be assured that the dynamic object will not be relocated (since the garbage collector will not be allowed to run) during execution of the system library routine.

Detailed Description Text (434):
All of real memory is divided into multiple fixed size segments of 512 Kbytes each. These segments are partitioned into a static region and a heap region. At run time, segments can be repartitioned.

Detailed Description Text (435):
1. The static region represents memory that is not relocated by the garbage collector. In general, this region comprises C stack segments for use by threads, segments of code produced by the JIT compiler, and stubs for byte-code and native methods.

Detailed Description Text (436):
2. The heap region comprises all of the remaining memory, which is divided into N equal-sized demispaces.

Detailed Description Text (438):
We intend for byte codes to be stored as part of the dynamic heap. This means they will be relocated as necessary on demand. However, the results of JIT compilation are stored in static memory. Note that each JIT-translated method is represented by a Method object which is stored in the garbage collected heap. The finalize() method for the Method object explicitly reclaims the static memory that had been reserved for representation of the method's JIT translation.

Detailed Description Text (439):
8.3 Global Pointer Variables (Roots)

Detailed Description Text (440):
All global root pointers must be registered so that they can be identified by the garbage collector at the start of each garbage collection pass. These root pointers are independently registered using the RegisterRooto macro, prototyped below. Each root pointer must be registered before its first use.

Detailed Description Text (443):
There are two static memory segments supported by our run-time system. Static memory segments are never relocated and are not currently garbage collected. The static data region represents the code produced by the JIT translator, native-method and byte-code-method stubs, and C stacks.

Detailed Description Text (448):
8.5 Heap Memory Allocation

Detailed Description Text (449):
This section describes the special techniques that are used to implement allocation

of objects within the garbage collected heap. Every newly allocated object can be assumed to contain all zeros.

Detailed Description Text (451):
Each heap-allocated object must be identified so that the garbage collector can determine which of its fields contain pointers. The standard technique for identifying pointers within heap objects is to provide a signature for each object. The signature pointer occupies a particular word of each object's header (See FIG. 2).

Detailed Description Text (453):
Within the signature structure, bitmap is an array of bits with one bit representing each word of the corresponding object. The bit has value zero if the corresponding word is a non-pointer, and value one if the corresponding word is a pointer. The first word of the object is represented by (bitmap[0]& 0.times.01). The second word is represented by (bitmap[0]& 0.times.02). The thirty-third word is represented by (bitmap[1]& 0.times.01), and so forth. Bits are provided only up to the word offset of the last pointer. Note that multiple heap-allocated objects may share the same statically allocated signature structure.

Detailed Description Text (454):
To simplify the creation of signatures, and to reduce the likelihood of programmer errors in specifying signatures, we provide a special C preprocessor that will automatically build signature declarations. The convention is to provide an appropriate preprocessor declaration to accompany each C structure that is defined. The following code fragment serves as an example:

Detailed Description Text (473):
The special preprocessor converts the signature macro to the following declaration: static int.sub.-- sig1234[ ]={5, Record .vertline.5, 0.times.01f,}; static struct Signature *.sub.-- sigClassFile=(struct Signature *).sub.-- sig1234; The codes used to identify fields within a structure are the same as the primitive C types: char, short, int, long, float, double. Note that we need not distinguish unsigned values. The ptr keyword represents pointers (the garbage collector does not need to know the type of the object pointed to).

Detailed Description Text (474):
In case of arrays, put the array dimension in square brackets immediately following the field specifier. For example:

Detailed Description Text (488):
Alternatively, programmers may refer to previously declared signatures by enclosing the structure name in angle braces (within the same preprocessor stream) as in the following:

Detailed Description Text (495):
Every PERC object begins with two special fields representing the object's lock and method tables respectively. See FIG. 23 for the declaration of MethodTable. The method table's first field is a pointer to the corresponding Class object. The second field is a pointer to an array of pointers to Method objects. The third field is a pointer to the JIT-code implementation of the first method, followed by a pointer to the JIT-code implementation of the second method, and so on. The pointers to JIT-code implementations may actually be pointers only to stub procedures that interface JIT code to byte-code or native-code methods.

Detailed Description Text (496):
Allocation routines. When allocating memory from within a native method, the programmer provides to the allocation routine the address of a signature rather than simply the size of the object to be allocated. The Signature pointer passed to each allocate routine must point to a statically allocated Signature structure. The

h e b b g e e e f c e e ge

implementation of the PERC virtual machine allocates a static signature for each class loaded. Once this static signature has been created, all subsequent instantiations of this class share access to this signature.

Detailed Description Text (500):
In some cases, such as when a dynamically allocated object contains union fields that contain pointers only some of the time, it is necessary to allocate a private copy of the signature along with the actual object. To minimize allocation overhead, both the signature and the data are allocated as a single contiguous region of memory using the following allocation routine, which assumes that its sp argument points to static memory:

Detailed Description Text (502):
If the signature itself must be dynamically constructed, use the following variant:

Detailed Description Text (521):
String and substring data is special in that we may have arrays of bytes that are shared by multiple overlapping strings. The bytes themselves are represented in a block of memory known to the garbage collector as a String. The programmer represents each string using a String object. FIG. 7 shows string objects x and y, representing the strings "embedded" and "bed" respectively. The value field of each string object is a pointer to the actual string data. The offset field is the offset, measured in bytes, of the start of the string within the corresponding StringData buffer. The count field is the number of bytes in the string. Note that count represents bytes, even though Unicode strings might require two bytes to represent each character.

Detailed Description Text (534):
8.6 Soft Real-Time Mostly Stationary Garbage Collection

Detailed Description Text (535):
This section describes the software implementation of a mostly stationary garbage collection technique. This represents the "best" stock-hardware compromise for reliable and fast execution within real-time constraints.

Detailed Description Text (536):
We use a mostly stationary garbage collection, in which memory is divided into 5 demispaces, each of size 512 Kbytes. At the start of each garbage collection, we select two regions to serve as to- and from-space respectively. All of the live objects currently residing in from-space are copied into to-space. At the end of garbage collection, the from-space region has been completely vacated of live memory, and thus consists of a large contiguous segment of free memory. One of the other three regions serves as a static region. It is excluded from the garbage collection process. The remaining two regions are garbage collected using an incremental mark and sweep technique. We identify the start of garbage collection as a flip.

Detailed Description Text (538):
At startup, flip as soon as 1/2 of memory has been allocated. Thereafter, flip as soon as the previous garbage collection pass completes. Use the following techniques and heuristics to allocate memory and select from-space:

Detailed Description Text (541):
3. Each object in memory is organized as illustrated in FIG. 2. The individual fields are as follows:

Detailed Description Text (542):
a. For objects residing in the mark-and-sweep region, the Scan List field distinguishes objects that have been marked from those that have not been marked.

At the start of garbage collection, every object's Scan List field has the NULL value, which is represented by the symbolic constant SCAN.sub.-- CLEAR. When an object is recognized as live, it is marked by inserting the object onto a list of objects needing to be scanned. This list is threaded through its Scan List field. To identify the last object on the scan list, its Scan List field is assigned the special value 0.times.01, which is represented by the symbolic constant SCAN.sub.-- END. For objects residing on a free list within the mark-and-sweep or to-space regions, the Scan List field has the special value 0.times.ffffffff, represented by the symbolic constant SCAN.sub.-- FREE.

Detailed Description Text (544):
Within to-space, the Scan List field is used to distinguish live objects from dead ones. Note that there are situations in which the same region might serve as to-space for two consecutive garbage collection passes. In this case, some of the objects residing in to-space at the start of garbage collection may actually be dead. At the start of garbage collection, all of the Scan List fields for objects residing in to-space are initialized to SCAN.sub.-- CLEAR. During garbage collection, any to-space object that is identified as live through scanning or normal application processing is placed onto the scan list (threaded through the Scan List field) if it had not previously been marked as live. For each object queued for copying into to-space, the Scan List field of the to-space copy is initialized to SCAN.sub.-- END to denote that the object is live.

Detailed Description Text (546):
c. Activity Pointer points to the real-time activity object that was responsible for allocation of this object or has the NULL value if this object was not allocated by a real-time activity. When this object's memory is reclaimed, that real-time activity's memory allocation budget will be increased. Furthermore, if this object needs to be finalized when the garbage collector endeavors to collect it, the object will be placed on a list of this real-time activity's objects which are awaiting finalization. To distinguish objects that need to be finalized, the 0.times.01 bit (FINAL.sub.-- LINK) and the 0.times.02 bit (FINAL.sub.-- OBJ) of the Activity Pointer field are set when a finalizable object is allocated.

Detailed Description Text (549):
5. At the time garbage collection begins (flip time), we sort the mark-and-sweep spaces according to amounts of available memory. Our preference is to allocate free memory from the space that is already most full. We link the free lists of the two mark-and-sweep free pools and the to-space free pool to reflect this preference. We always put to-space as the last region on this list, because we prefer to allocate from the mark-and-sweep regions if they have space available to us.

Detailed Description Text (550):
6. To allocate a heap object from a region's free pool, select the first (smallest) free list that is known to contain free segments of sufficiently large size. If the free list is not empty, remove the first segment on that free list, divide that segment into two smaller segments with one being of the requested size and the other being returned to the appropriate free list (if the free segment is sufficiently large), and return the allocated memory. If the selected free list is empty, repeat this algorithm on the next larger size free list (until there are no larger free lists to try).

Detailed Description Text (553):
In Java, programmers can specify an action to be performed when objects of certain types are reclaimed by the garbage collector. These actions are specified by including a non-empty finalize method in the class definition. Such objects are said to be finalizable. When a finalizable object is allocated, the two low order bits of the Activity Pointer are set to indicate that the object is finalizable. The 0.times.01 bit, known symbolically as FINAL.sub.-- LINK, signifies that this object has an extra Finalize Link field appended to the end of it. The 0.times.02

bit, known symbolically as FINAL.sub.-- OBJ, signifies that this object needs to be finalized. After the object has been finalized once, its FINAL.sub.-- OBJ is cleared, but its FINAL.sub.-- LIINK bit remains on throughout the object's lifetime.

Detailed Description Text (554):
See FIG. 3 for an illustration of how finalization lists are organized. In this figure, Finalizees is a root pointer. This pointer refers to a list of finalization-list headers. There is one such list for each of the currently executing real-time activities, and there is one other list that represents all of the objects allocated by non-real-time activities. These lists are linked through the Activity Pointer field of the objects waiting to be finalized.

Detailed Description Text (555):
The run-time system includes a background finalizer thread which takes responsibility for incrementally executing the finalizers associated with all of the objects reachable from the Finalizees root pointer. Following execution of the finalizer method, the finalizes object is removed from the finalizes list and its Activity Pointer field is overwritten with a reference to the corresponding Activity object. Furthermore, we clear the FINAL.sub.-- OBJ so we don't finalize it again. Optionally, each real-time activity may take responsibility for timely finalization of its own finalizee objects. Typically, this is done within an ongoing real-time thread that is part of the activity's workload.

Detailed Description Text (558):
8.6.3 Synchronization Between Application Code and Incremental Garbage Collection

Detailed Description Text (559):
Garbage collection is performed as an incremental background process. Application code honors the following protocols in order to not interfere with background garbage collection activities.

Detailed Description Text (560):
1. Heap memory that has already been examined by the garbage collector must not be corrupted by writing into such heap objects pointers that have not yet been processed by the garbage collector. Otherwise, it might be possible for a pointer to escape scrutiny of the garbage collector. As a result, the referenced object might be treated as garbage and accidentally reclaimed. To avoid this problem, we impose a write barrier whenever pointers are written into the heap. (See the SetHeapPointer() macro in FIG. 41):

Detailed Description Text (563):
2. We do not impose a read barrier. This means that pointers fetched from the internal fields of heap objects may refer to from-space objects or to mark-and-sweep objects that have not yet been marked. In case a pointer refers to a from-space object that has already been copied into to-space or to a to-space object that has not yet been copied into to-space, all references to heap object are indirected through the Indirection Pointer. (See FIG. 41)

Detailed Description Text (564):
3. Any objects that are newly allocated from the mark-and-sweep region have their Scan List pointer initialized to NULL. Thus, newly allocated objects will survive the current garbage collection pass only if pointers to these objects are written into the live heap.

Detailed Description Text (565):
There are two garbage collection techniques being carried out in parallel: copying between from- and to-space, and incremental mark-and-sweep in the remaining regions. Garbage collection begins with identification of the live objects that are referenced from the root registers. The flip operation consists, therefore, of the

following actions:

Detailed Description Text (566):
1. The garbage collector sorts heap regions in descending order according to amount of allocated memory. The last region on this sorted list is known to be completely free, since at least from-space, and possibly other regions, is known to have serviced no allocation requests during the most recent garbage collection pass.

Detailed Description Text (569):
c. Divide the available memory in the newly selected to-space into one segment for allocation of new memory requests and another segment for copying of live from-spacc objects. The region reserved for copying is assigned to lower addresses, and is large enough to hold all of the memory currently allocated in from-space, even though some of the from-space objects are likely to be dead and will not need to be copied. From-space objects are copied into to-space from low to high address. New memory is allocated within to-space from high to low address. At the end of garbage collection, we coalesce whatever is left over from the copy region with whatever is left from the allocate region into a single contiguous segment of free memory.

Detailed Description Text (571):
2. Tend each root pointer. This consists of:

Detailed Description Text (572):
a. If the pointer refers to from-space, allocate space for a copy of this object in to-space and make the to-space copy's Indirect Pointer refer to the from-space object. Set the root pointer to refer to the to-space copy. Set the from-space copy's Scan List pointer to refer to the to-space copy.

Detailed Description Text (573):
b. Otherwise, if the pointer refers to the mark-and-sweep region or the to-space region and the referenced object has not yet been marked, mark the object. Marking consists of placing the object on the scan list. Each increment of garbage collection effort consists of the following:

Detailed Description Text (574):
1. If we are not searching for objects in need of finalization and if there is garbage collection work to be done for the copy region, do it.

Detailed Description Text (578):
a. Rescan the root registers.

Detailed Description Text (579):
b. Following the root register scan, if there is no more memory to be scanned and there is no more memory to be copied, consider the mark process to have been terminated. Our next job is to search for finalizable objects.

Detailed Description Text (583):
7. Else, do a flip operation and restart the garbage collector. To-space and from-space are organized as illustrated in FIG. 4. In this illustration, live objects A, B, and C are being copied into to-space out of from-space. Objects B and C have been copied and object A is on the copy queue waiting to be copied. The arrows indicate the values of the Indirect Pointer fields in each of the invalid object copies. Memory to the right of the New pointer consists of objects that have been allocated during this pass of the garbage collector. Memory to the left of B' represents objects that were copied to to-space during the previous pass of the garbage collector. Garbage collection of the copy region consists of the following:

Detailed Description Text (587):
Additionally, tend the Activity Pointer field after masking out its two least

significant bits and update the Signature Pointer if the signature is contained within this object. Scanning of the mark-and-sweep region consists of the following:

Detailed Description Text (589):
a. Scan the object at the head of the list. Scanning consists of tending each pointer contained within the object. Note that the scanner must scan the Activity Pointer field (after masking out the two least significant bits). A special technique is used to scan pointer stack objects. When pointer stack objects are scanned, the garbage collector consults the corresponding Thread object to determine the current height of the pointer stack. Rather than scan the entire object, the garbage collector only scans that portion of the stack object that is currently being used.

Detailed Description Text (591):
Scanning of PERC pointer stacks is special in the sense that only the portion of the stack that is live is scanned. Memory within the object that is above the top-of-stack pointer is ignored. In order to support this capability, PERC pointer stacks refer to their corresponding Thread object, enabling the garbage collector to consult the thread's top-of-stack pointer before scanning the stack object.

Detailed Description Text (592):
Once we are done with the marking and copying process, our next responsibility is to search for objects in need of finalization. The search process consists of the following steps:

Detailed Description Text (598):
3. Wait for the scanning and copying process to complete. It is not necessary to rescan the root pointers because all of the objects now being scanned and copied are considered to be dead insofar as the application code is concerned. Thus, there is no possible way for a pointer to one of these "dead" objects to find its way into a root pointer.

Detailed Description Text (600):
Next, we sweep the entire mark-and-sweep and to-space regions. Before sweeping to-space, we coalesce the unused portion of the memory segment which had been reserved for copying of from-space objects with the free segment that immediately follows this segment. Sweeping consists of the following steps:

Detailed Description Text (604):
c. If it is not marked and is not already on a free list, we have discovered a candidate for reclamation. Check to see if this is a HashLock object. If so, the garbage collector first reclaims the hash value by (i) checking to see if there is an available slot in the hash-value manager's list of recycled hash values and copying this object's hash value into that slot if so, or (ii) making this object live, changing its signature to that of a HashCache object, and linking the HashCache object onto the hash-value manager's list of recycled hash values.

Detailed Description Text (605):
d. Assuming this object has not been converted into a HashCache object, we place this object onto the appropriate free list after first merging with the preceding object if the preceding object is also free. In the process of reclaiming this object's memory, update the corresponding activity's tally that represents the total amount of this activity's previously allocated memory that has been garbage collected. Also, zero out all of the memory contained within the newly reclaimed object.

Detailed Description Text (606):
The final step is to zero out the old from-space so that future allocations from this region can be assumed to contain only zeros. Simply walk through memory from

low to high address and overwrite each word with a zero. For each object encountered in from-space, we ask whether it was copied into to-space (by examining its Indirect Pointer). If it was not copied, we check to see if it is a HashLock object with a hash value that needs to be reclaimed. If so, we reclaim the hash value as described above, except that a new HashCache object may need to be allocated to represent the recycled hash value if there are no available slots in the existing list of recycled hash values. We allocate this HashCache object using the standard heap-memory allocator. Otherwise, we update the corresponding activity's tally that represents the total amount of this activity's previously allocated memory that has been garbage collected.

Detailed Description Text (608):
The standard model for execution of Java byte-code programs assumes an execution model comprised of a single stack. Furthermore, the Java byte codes are designed to support dynamic loading and linking. This requires the use of symbolic references to external symbols. Resolving these symbolic references is a fairly costly operation which should not be performed each time an external reference is accessed. Instead, the PERC virtual machine replaces symbolic references with more efficient integer index and direct pointer references when the code is loaded.

Detailed Description Text (609):
In order to achieve good performance, the PERC virtual machine does not check for type correctness of arguments each time it executes a byte-code instruction. Rather, it assumes that the supplied arguments are of the appropriate type. Since byte-code programs may be downloaded from remote computer systems, some of which are not necessarily trustworthy, it is necessary for the PERC virtual machine to scrutinize the byte-code program for type correctness before it begins to execute. The process of guaranteeing that all of the operands supplied to each byte-code instruction are of the appropriate type is known as byte code verification. Once the types of each operation are known, it is possible to perform certain code transformations. Some of these transformations are designed simply to improve performance. In other cases, the transformations are needed to comply with the special requirements of the PERC virtual machine's stack protocols. For example, Java's dup2 byte code duplicates the top two elements on the Java stack. Byte-code verification determines the types of the top two stack elements. If both are of type pointer, the class loader replaces this byte code with a special instruction named dup2.sub.-- 11, which duplicates the top two elements of the pointer stack. If the two stack arguments are both non-pointer values, the PERC class loader replaces this byte code with the dup2.sub.-- 00 instruction, which duplicates the top two elements of the non-pointer stack. If one of dup's stack arguments is a pointer and the other is a non-pointer (in either order), the PERC class loader replaces dup with dup2.sub.-- 10, which duplicates the top element on each stack. A complete list of all the transformations that are performed by the byte code loader is provided in the remainder of this section.

Detailed Description Text (613):
Before starting the second pass, we identify each of the entry points to the method. We consider the first basic block in the method to be the main entry point. Additionally, we consider the starting block for each finally statement to represent an entry point. And further, we consider the starting block for each exception handler to represent an entry point. Exception handlers are identified in the method's code attribute, in the field named exception.sub.-- table. The relevant data structures are described in "The Java Virtual Machine Specification", written by T im Lindholm and Frank Yellin, published in 1996.

Detailed Description Text (614):
Each basic block is represented by a data structure with fields representing the following information:

Detailed Description Text (623):

h e b b g e e e f c e e ge

Consider analysis of the main entry point and the blocks reachable from this entry point. First, we initialize the entry point's initial stack to empty. Then we simulate execution of the entry block's instructions and record the effects of these instructions in terms of the types of the values that will be popped from and pushed onto the stack. After simulating all of the instructions in this basic block, we examine each of the entry block's successors as follows:

Detailed Description Text (630):
Most of the operations that access the constant pool can be replaced with fast variants. When a Java class is loaded into the Java virtual machine, all of the constants associated with each method are loaded into a data structure known as the constant pool. Because Java programs are linked together at run time, many constants are represented symbolically in the byte code. Once the program has been loaded, the symbolic values are replaced in the constant pool with the actual constants they represent. We call this process "resolving constants." Sun Microsystems Inc.'s descriptions of their Java implementation suggest that constants should be resolved on the fly: each constant is resolved the first time it is accessed by user code. Sun Microsystems Inc.'s documents further suggest that once an instruction making reference to a constant value has been executed and the corresponding constant has been resolved, that byte code instruction should be replaced with a quick variant of the same instruction. The main difference between the quick variant and the original instruction is that the quick variant knows that the corresponding constant has already been resolved.

Detailed Description Text (634):
Putfield. This operation is represented by code 181. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. The loader replaces this code with one of the following:

Detailed Description Text (635):
1. putfield.sub.-- q encoded as 181: We replace the constant-pool entry with an integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less than 64 Kbytes and can thus be represented in the 16-bit immediate operand representing an unsigned integer quantity.

Detailed Description Text (641):
Getfield. This operation is represented by code 180. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. This code is replaced with one of the following:

Detailed Description Text (642):
1. getfield.sub.-- q encoded as 180: We replace the constant-pool entry with a 32-bit integer that represents the field's offset, size, and tag to indicate whether the field contains a pointer. This information is encoded such that the most significant bit is on if the field contains a pointer, the next two bits encode the size of the field, and the remaining 29 bits represent the field's offset. The constant-pool entry is tagged so that other putfield and getfield operations that refer to the same constant-pool entry can be appropriately resolved. Only use this instruction if the field offset is larger than can be represented in 16 unsigned bits. The instructions that follow handle cases in which the field offset is less

than 64 Kbytes and can thus be represented in the 16-bit immediate operand representing an unsigned integer quantity.

Detailed Description Text (648):
Putstatic. This operation is represented by code 179. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a pointer to the Field structure that describes the field to be updated. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (654):
Getstatic. This operation is represented by code 178. It takes a two-byte immediate operand which represents an index into the constant pool. This index indirectly represents the offset of the field within the corresponding object and the width of the field, measured in bits. We replace the selected constant-pool entry with a pointer to the Field structure that describes the field to be fetched. This field structure includes a pointer to the corresponding class object and also includes the offset of the field within the class object. This code is replaced with one of the following:

Detailed Description Text (668):
When byte codes are processed by the ROMizer tool for placement in ROM memory, the invokeinterface instruction is replaced with invokeinterface.sub.-- qrom, encoded as 216. This instruction is distinguished from invokeinterface.sub.-- q only in that the reserved operand is an index into a 256-element array of guesses maintained by the PERC virtual machine for the purpose of supporting customization of invokeinterface instructions. If the ROMizer's output contains fewer than 256 invokeinterface.sub.-- qrom instructions, then each one's reserved operand will have a different integer value in the range 0 to 255 inclusive. Otherwise, certain invokeinterface.sub.-- qrom instructions will share access to the same slot in the guess array.

Detailed Description Text (674):
There is one special context in which astore and astore.sub.-- <n> instructions require special handling. In the code generated for the body of a finally statement, javac uses an astore instruction to store the return address. The PERC virtual machine treats the return address as an integer, and thus replaces this astore instruction with an istore.

Detailed Description Text (738):
The output of the ROMizer tool is a load file designed to be burned into a ROM. This load file is organized as follows:

Detailed Description Text (740):
1. All objects placed into the Object.sub.-- Region are marked by setting their Scan List field to SCAN.sub.-- END. This prevents on-the-fly write barrier enforcement from attempting to place these objects on the scan list.

Detailed Description Text (741):
2. All Indirect Pointers are initialized to refer to the object itself. This enables standard heap-access macros to work correctly when referring to ROM objects.

Detailed Description Text (748):
An important consideration in the architecture of a large software system such as the PFERC virtual machine is the need to minimize the effort required to implement and maintain the various capabilities of the system. There are several innovations

represented in the design of our ROMizer tool:

Detailed Description Text (749):
1. The code used in the implementation of the ROMizer tool to read in a Java class file, verify the validity of the byte code, and transform the byte code into the PERC instruction set is the exact same code that is used by the PERC implementation to support dynamic (on-the-fly) loading of new byte-code functionality into the PERC virtual machine.

Other Reference Publication (6):
Evans et al, "Garbage collection and memeory management", ACM 138-143, 1997.

CLAIMS:

1. A real-time virtual machine method (RTVMM) for implementing real-time systems and activities, the RTVMM comprising the steps:

implementing an O-OPL program that can run on computer systems of different designs, an O-OPL program being based on an object-oriented programming language (O-OPL) comprising object type declarations called classes, each class definition describing the variables that are associated with each object of the corresponding class and all of the operations called methods that can be applied to instantiated objects of the specified type, a "method" being a term of art describing the unit of procedural abstraction in an object-oriented programming system, an O-OPL program comprising one or more threads wherein the run-time stack for each thread is organized so as to allow accurate identification of type-tagged pointers contained on the stack without requiring type tag information to be updated each time the stack's content changes, the O-OPL being an extension of a high-level language (HLL) exemplified by Java, HLL being an extension of a low-level language (LLL) exemplified by C and C++, a thread being a term of art for an independently-executing task, an O-OPL program being represented at run time by either O-OPL byte codes or by native machine codes.

3. The RTVMM of claim 1 wherein an O-OPL program comprises one or more classes represented in read-only memory, the methods thereof h aving been converted into O-OPL byte codes prior to run time.

4. The RTVMM of claim 1 wherein an O-OPL program comprises one or more classes represented in read-only memory, the methods thereof having been converted into native machine language prior to run time.

11. The RTVMM of claim 1 wherein the implementing step comprises the step:

causing an application thread that is to be preempted to provide notification as to when the thread is at a point where safe garbage collection can take place.

12. The RTVMM of claim 1 wherein one of the implemented threads is a garbage collection thread that operates asynchronously thereby resulting in the garbage collection thread being interleaved with other threads in arbitrary order, objects subject to garbage collection being either finalizable or non-finalizable, a finalizable object being subject to an action that is performed when the memory space allocated to the finalizable object is reclaimed by the garbage collection thread, the finalizing action being specified by including a non-empty finalizer method in the class definition, the garbage collection thread being able to distinguish a thread's pointer variables from the thread's non-pointer variables, preemption of a thread being allowed only if the thread is in a state identified as a preemption point, a thread being allowed to hold pointers in variables between preemption points that may not be visible to the garbage collection thread, pointer variables that may not be visible to the garbage collection thread being called fast pointers, pointer variables that are visible to the garbage collection thread

being called slow pointers, each LLL, function being identified as either preemptible or non-preemptible.

15. The RTVMM of claim 12 wherein the implementing step comprises the steps:

providing a plurality of macros representing (1) an interface that permits the use of different garbage-collection techniques and (2) an implementation of a mostly-stationary garbage-collection technique.

16. The RTVMM of claim 12 wherein the implementing step comprises the step:

providing parameterized access to heap memory in order to facilitate the implementation of read and write barriers, heap memory being a region of memory wherein objects of arbitrary size can be allocated space to satisfy the dynamic memory needs of application programs, heap memory being subject to garbage collection.

17. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that returns the value of a fast pointer in the heap given the identity of the pointer and its type.

18. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that assigns a value from a fast pointer in heap memory given the identity of the pointer, its type, and the value.

19. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that returns the value of a nonpointer in heap memory given the identity of the nonpointer and its type.

20. The RTVMM of claim 16 wherein the implementing step comprises the step:

providing a macro that assigns a value to a nonpointer in heap memory given the identity of the nonpointer, its type, and the value.

24. The RTVMM of claim 12 wherein the implementing step comprises the steps:

partitioning memory into at least three demi-spaces, at least one of the demi-spaces being a static space excluded from the garbage collection process;

designating two of the demi-spaces as to-space and from-space at the beginning of a garbage collection cycle, live objects residing in from-space subsequently being copied into to-space;

designating the remaining demi-spaces as mark-and-sweep spaces at the beginning of a garbage collection cycle, the mark-and-sweep spaces being garbage collected using a mark-and-sweep technique.

29. The RTVMM of claim 24 wherein a garbage-collection cycle begins, the implementing step comprising the steps:

causing the non-empty mark-and-sweep space having the most available free space to be designated as the new from-space;

causing the old to-space to be designated as the new to-space if the allocated space within the new from-space is less than the free space available as a single contiguous region in the old to-space; otherwise,

causing the old from-space to be designated as the new to-space.

30. The RTVMM of claim 24 wherein the implementing step comprises the step:

including a "scan list" field for each object in memory, the "scan list" field distinguishing marked and unmarked objects residing in a mark-and-sweep space but not on a free list, the "scan list" field for each object in a mark-and-sweep space having a "scan clear" value at the beginning of a garbage collection cycle, an object recognized as being a live object being placed on a list of recognized live objects, the "scan list" field for an object on the list of recognized live objects having either a "scan end" value denoting the last object on the list of recognized live objects or a value identifying the next object on the list of recognized live objects, the "scan list" field for an object residing on a free list within a mark-and-sweep space or to- space having the "scan free" value, the "scan list" field for an object residing in from-space which has been scheduled for copying into to-space being a pointer to the to-space copy, the "scan list" field otherwise being assigned the "scan clear" value, the "scan list" field for an object residing in to-space having the "scan clear" value at the beginning of a garbage collection cycle, a to-space object recognized as live during garbage collection being placed on a list of recognized live objects, the "scan list" field for a to-space object on the list of recognized live objects having a value identifying the next object on the list of recognized live objects, the "scan list" field for each object queued for copying into to-space having the "scan end" value denoting that the object is live.

31. The RTVMM of claim 24 wherein the implementing step comprises the steps:

providing a memory allocation budget for each real-time activity;

allocating memory from the memory allocation budget to an object associated with the real-time activity;

causing the garbage collection thread to credit the memory allocation budget of the real-time activity when the memory allocated to the object is reclaimed.

32. The RTVMM of claim 24 wherein a real-time activity has allocated memory to an object which is subject to finalization and the garbage collection thread endeavors to reclaim the allocated memory, the implementing step comprising the step:

causing the garbage collection thread to place the object on a list of the real-time activity's objects that are awaiting finalization.

36. The RTVMM of claim 24 wherein the implementing step comprises the steps:

causing the available memory in a newly-selected to-space to be divided into a new-object segment for allocation of memory to new objects and an old-object segment for receiving copies of live from-space objects, the old-object segment being equal to or larger than the allocated space in from-space, new objects being allocated space in sequence from the end of the new-object segment away from the old-object segment, old objects being copied in sequence from the end of the old-object segment away from the new-object segment;

causing the unallocated portions of the old-object segment and the new-object segment to be coalesced into a single contiguous segment of free memory at the end of a garbage collection cycle.

37. The RTVMM of claim 24 wherein, after to-space and from-space have been selected at the beginning of a garbage collection cycle, the implementing step comprises the steps:

causing the free pools of memory in the mark-and-sweep spaces and to-space to be linked together into a global free pool, the free pools of the mark-and-sweep spaces being linked in increasing order of amount of free memory, the free pool of to-space being linked to the mark-and-sweep space having the greatest amount of free memory, a request for a new memory allocation being satisfied by the first memory segment of sufficient size found by searching the global free pool according to the linking order.

38. The RTVMM of claim 24 wherein the implementing step comprises the steps:

maintaining a list of root pointers to live objects;

causing space for a copy of an object in to-space to be allocated if a root pointer to the object refers to from-space;

causing the from-space address of the object to be written in an "indirect pointer" field of the object's allocated space in to-space;

causing the root pointer to be replaced with the address of the object in to-space;

causing the to-space address of the object to be written into a "scan list" field of the object in from-space.

39. The RTVMM of claim 24 wherein the implementing step comprises the steps:

maintaining a list of root pointers to live objects;

causing an object to be marked if the root pointer to the object refers to a mark-and-sweep space or to-space and the object has not yet been marked, marking consisting of placing the object on a scan list.

40. The RTVMM of claim 24 wherein the marking and copying processes for a particular garbage collection cycle have been completed, the implementing step comprising the steps:

causing all objects needing finalization to be transferred from a list of finalizable objects to a finalizee list;

causing the transferred objects residing in mark-and-sweep space to be placed on a scan list;

causing the transferred objects residing in from-space to be placed on a copy list.

41. The RTVMM of claim 40 wherein the marking and copying processes for a particular garbage collection cycle have been completed, the implementing step comprising the steps:

causing an object from a list of finalizable objects to be transferred to a finalizee list if the object has not been marked or if the object is a from-space object that has not been copied into to-space, the object being placed on the copy list and space being allocated in to-space if the object resides in from-space, the object being marked by being placed on the scan list if the object resides in mark-and-sweep space.

43. The RTVMM of claim 40 wherein the transfer of objects needing finalization on the list of finalizable objects to the finalizee list has been completed, the implementing step comprising the steps:

causing the objects on the copy list to be copied to to-space;

causing the objects on the scan list to be scanned, scanning consisting of tending each pointer contained within an object, tending being a term of art describing the garbage collection process of (1) examining a pointer and, if the object has not already been recognized as live, arranging for the referenced object to be subsequently scanned by placing the object on a scan list if it resides in a mark-and-sweep space or in to-space or by arranging for the object to be copied into to-space if it resides in from-space and (2) updating the pointer to refer to the object's new location if it has been queued for copying into to-space.

47. The RTVMM of claim 24 wherein the transfer of objects needing finalization from a list of finalizable objects to an activity's finalizee list or an orphaned finalizee list has been accomplished, the implementing step comprising the steps:

causing the mark-and-sweep spaces and to-space to be swept and identifying each object that is not marked, that is not on a free list, and that is a "hashlock object";

causing the garbage collection thread to copy the value of a "hash value" field of the "hashlock object" onto a list of recycled hash values if the list is not full; otherwise:

causing the garbage collection thread to (1) make the "hashlock object" live, (2) change a "signature" field in the "hashlock object" to represent a "hashcache object", (3) add the "hashcache object" to the list of recycled hash values, and (4) copy the value of the "hash value" field of the original "hashlock object" onto a list of recycled hash values.

48. The RTVMM of claim 24 wherein the transfer of objects needing finalization from a list of finalizable objects to an activity's finalizee list or an orphaned finalizee list has been accomplished, the implementing step comprising the steps:

causing from-space to be examined and each object to be identified that was not copied into to-space and that is a "hashlock object" with a hash value that needs to be reclaimed;

causing the garbage collection thread to copy the value of a "hash value" field of the "hashlock object" into a list of recycled hash values if the list is not full; otherwise:

causing the garbage collection thread to (1) make the "hashlock object" live, (2) change a "signature" field in the "hashlock object" to represent a "hashcache object", (3) add the "hashcache object" to the list of recycled hash values, and (4) copy the value of the "hash value" field of the original "hashlock object" onto a list of recycled hash values;

causing zeros to be written into all of from-space.

53. The RTVMM of claim 12 wherein the implementing step comprises the step:

including an "activity pointer" field for each object in memory, the "activity pointer" identifying the real-time activity object that was responsible for allocation of the object, the "activity pointer" field containing a "null" value if the object was not allocated by a real-time activity;

maintaining a finalizees list of objects waiting to be finalized for each real-time activity, the objects on the finalizees list being linked through the "activity pointer" field;

maintaining a list of the headers of the finalizees lists, the pointer "finalizees" being a root pointer to the headers list.

61. The RTVMM of claim 59 wherein one of the implemented threads is a garbage collection thread the implementing step comprising the steps:

maintaining a list of available hash values consisting of previously assigned hash values for which the corresponding objects have been reclaimed by the garbage collection thread;

causing one of the hash values on the list of available hash values to be designated as the next available hash value to be assigned to a "hash object" if the list of available hash values is non-empty;

causing a static counter to be incremented if the list of available hash values is empty and causing the new counter value to be designated as the next available hash value to be assigned to a "hash object".